Evaluating Self-Correcting LLM Agents for Robust Test Assertion Generation

Thesis Information

Supervisor(s): Annibale Panichella, Mitchell Olsthoorn
A Thesis Submitted to EEMCS Faculty Delft University of Technology, In Partial Fulfilment of the Requirements For the Bachelor of Computer Science and Engineering
June 20, 2026
Final project course: CSE3000 Research Project
Thesis committee: Annibale Panichella, Mitchell Olsthoorn, Alex Voulimeneas

An electronic version of this thesis is available at https://github.com/horiagali/assertion-generator-java

Horia Galitianu

EEMCS, Delft University of Technology
The Netherlands

Abstract

Robust test assertions are critical for verifying deep semantic behavior, but their automated generation remains a primary bottleneck in software testing. Automated test case generation approaches often rely on implicit oracles or regression checks that miss semantic failures. Large Language Models can synthesize meaningful assertions, but single-pass prompting frequently produces uncompilable or failing code. We propose a multi-agent workflow for Java test assertion generation consisting of code comprehension, test objective planning, and assertion generation. The workflow extracts mutation-relevant variable manifests, structures high-level testing plans, compiles and executes the generated test candidates, and iteratively refines assertions using mutation testing feedback from PITest to optimize mutation quality before final selection. We evaluate the approach on 112 focal tests from twilio-java and liqp. Compared with static prompting, agentic configurations substantially improve reliability, increasing the percentage of valid runs (compilable, executing and passing tests) from 58.1% to 84.8%. Relative to the human baseline, the agentic configuration raises the average Test Strength (the ratio of killed mutants strictly to covered mutants) from 45.6% to approximately 56%. Our evaluation finds that while execution feedback significantly improves reliability and observed Test Strength, combining all agentic components does not yield the best computational trade-off.

1 Introduction

A central limitation in automated software testing is the difficulty of generating reliable test oracles [2]. A test case consists of input data, method calls, and a test oracle (assertions). While automated test case generation approaches successfully automate full test cases using search-based techniques, they rely heavily on implicit oracles or regression checks that merely capture current, potentially buggy, behavior. Consequently, these generated tests must be manually verified by developers to determine the correct behavior, which can be time-intensive.

Large Language Models (LLMs) offer a promising solution by synthesizing semantically meaningful assertions from source code context. Because input generation is already handled well by existing fuzzing and search-based tools, we focus purely on the assertion generation bottleneck. Recent frameworks such as Nexus [7] and CANDOR [13] explore execution-grounded and multi-agent approaches to validating candidate oracles. However, many assertion-generation pipelines still rely on static, single-pass prompting. As Chu et al. [4] highlight in a systematic review, single-pass LLM generation frequently suffers from low usability, high error rates, and compilation issues. Because static models do not compile their own outputs or inspect execution feedback, a substantial fraction of generated oracles can fail before mutation testing can evaluate their semantic value. Empirical studies confirm that relying on a single prompt without automated feedback mechanisms can reduce the pass rate and robustness of generated tests [12].

While early iterative frameworks show promise, we identify a significant gap in the literature regarding the systematic evaluation of agentic architectures for assertion-only oracle generation. Specifically, current research lacks ablation studies that isolate and quantify the impact of individual workflow components, such as pre-generation planning, context summarization, and mutation-driven feedback, on compilability, execution reliability, and final assertion quality.

To address these limitations, we introduce a multi-agent workflow for automated test oracle generation and systematically evaluate its mechanisms of self-correction. The system extracts mutation-relevant variables, maps out testing strategies, generates code, and evaluates the results directly against a test runner and a mutation testing engine (PITest). If an assertion fails to compile or leaves mutants alive, the resulting error traces are fed back to an improvement agent that fixes the code in the subsequent iteration.

The main contributions of this paper are:

- A multi-agent framework tailored specifically for Java test assertion generation using compile-time and mutation feedback.

- An extensive ablation study isolating the impacts of summarization, planning, and iterative refinement.
- Empirical evidence demonstrating that execution feedback is the primary driver of reliability improvements over static prompting.

2 Background and Related Work

2.1 Test Generation and Oracles

Prior to the adoption of LLMs, automated generation relied heavily on Search-Based Software Testing (SBST) tools like EvoSuite [6]. These tools generate high-coverage test inputs but generally produce weak implicit oracles (e.g., checking for unexpected crashes) or regression assertions that lock in existing behavior. Evaluating the semantic quality of these oracles is done using mutation testing, a fault-injection technique [1]. We operate this technique by introducing small, artificial syntactic changes into the program source code.

To specifically evaluate assertion quality, we utilize test strength, dividing the number of killed mutants by the number of covered mutants. This isolates the quality of the assertion from the input execution path. We best understand the necessity of test strength through the RIP model of fault observation [1]:

(1) **Reachability:** The test execution must reach the mutated statement.
(2) **Infection:** The mutant must alter the internal state of the program.
(3) **Propagation (Observability):** The infected state must propagate to an output that the test assertion can observe and reject.

If a mutant remains unreachable, an assertion cannot kill it. By focusing solely on test strength, we evaluate the LLM exclusively on its ability to ensure propagation and observability for the code paths it actually reaches.

2.2 LLM-Based Test Oracle Generation

Recent research explores LLMs for generating robust test oracles to overcome rule-based limitations. Molinelli et al. evaluated LLM-generated oracles on real-world Java projects [9], showing LLMs can write effective assertions but suffer from high failure rates in single-pass setups. Other recent works cluster into specific optimization strategies. For example, AugmenTest leverages LLMs to infer correct test oracles based strictly on available documentation and developer comments rather than the code itself [8]. Similarly, AsserT5 employs a fine-tuned CodeT5 model to predict suitable test assertions for specific test scenarios, demonstrating that abstraction and the inclusion of focal methods improve precision [11].

Tratto uses a neuro-symbolic approach to constrain oracle generation [10]. Recent iterative frameworks such as LLMLOOP and GEM use execution feedback, static analysis feedback, and mutation feedback to repair or strengthen generated code and tests [3, 12].

An agentic workflow transitions an LLM into a system that interacts with external tools. Despite advancements in this area, the specific contribution of individual agentic components to assertion-only oracle generation remains under-explored. We formalized these specific steps to isolate how pre-generation summarization, planning, and execution-driven refinement independently affect final assertion quality.

3 Approach

We propose a multi-agent workflow consisting of code comprehension, test objective planning, and assertion generation. We implement this framework using LangGraph to manage state and control flow. To overcome the limitations of static prompting, we construct a system that generates assertions, executes them against a mutation testing engine, and uses the resulting feedback to improve the candidate assertions. We assign separate responsibilities to distinct reasoning nodes to isolate performance bottlenecks.

Figure 1 shows the full graph architecture and the feedback loop used during refinement.

3.1 Workflow Components

We provide the system prompt for each node in Appendix A.

3.1.1 Summarizer. We condense the full prompt context into a compact assertion-oriented summary. The agent preserves the test prefix state, visible variables, setup/helper methods, focal-class APIs, return types, and mutation-relevant behavior needed for assertion generation. The objective is to extract only the information required to generate high-quality mutation-killing assertions without hallucinating hidden state.

3.1.2 Planner. We prompt the agent to translate the summarized manifest into high-level strategic assertion goals. We restrict this node from writing any Java code, forcing it to focus purely on testing strategy. The objective is to map out assertion candidates and evaluate mutation risks before the coding phase begins.

3.1.3 Coder. We instruct the LLM to synthesize the available context and plan into pure Java test-body statements. The agent generates only raw Java statements and JUnit-style assertions, without classes, imports, annotations, comments, or helper methods.

3.1.4 Critic. This node is a deterministic script, not an LLM. We inject the Coder's candidate assertions into the actual Java test file within a controlled sandbox environment. We compile the code and execute PITest [5]. If compilation fails, the error log is captured. The Critic node records the test strength and extracts the precise, line-by-line list of surviving mutants or compilation errors.

3.1.5 Improver. We feed the execution logs from the Critic back to the LLM. The agent is explicitly prompted to analyze why specific mutants survived or why the code failed to compile, producing a targeted refinement plan to guide the Coder in the subsequent iteration.

3.2 Conditional Routing and Termination

We control the cyclic flow of the graph using a routing function positioned immediately after the Critic node. We terminate the loop and return the best-performing assertion set when the graph meets one of three conditions:

(1) The agent achieves a perfect test strength of 1.0.

Figure 1: Architecture of the self-correcting agentic oracle-generation workflow.

(2) The loop reaches the predefined maximum of four iterations, chosen as an empirical balance between computational cost and generation improvement.
(3) The Critic returns the exact same survival feedback as the previous iteration.

4 Study Design

To systematically evaluate the capabilities of our agentic workflow, we establish a reproducible experimental pipeline comparing iterative generation against static prompting baselines.

4.1 Research Question

To guide this evaluation, we address the following primary question:

> RQ: To what extent does an iterative, multi-agent framework improve the compilation, execution, and Test Strength of LLM-generated test assertions compared to static, one-shot prompting?

We conduct a structured ablation study. We isolate the impact of individual architectural components to systematically break down their specific contributions to overall execution reliability and final Test Strength.

4.2 Data Collection

We construct our dataset from two real-world Java projects, twilio-java and liqp, utilizing them for their varying complexity and well-structured unit tests. The raw dataset contains 112 JSON datapoints. A datapoint represents a single test prefix (the arrangement and action phases of a test).

4.3 Model and Configuration

Both the static baseline and the agentic configurations use the same LLM: `qwen3-coder:480b-cloud`. We use deterministic decoding with temperature 0.

4.4 Experimental Pipeline

(1) Prefix Preparation: We strip all human-written assertions from existing tests, leaving only the arrangement and action phases.
(2) Oracle Injection: For each test prefix, we execute our workflow across the defined ablation configurations and inject the generated assertions back into the test class placeholder.
(3) Execution and Mutation Analysis: We compile the injected test classes within an isolated sandbox. If the suite compiles and passes the initial execution, we execute PITest to perform mutation analysis.

4.5 Evaluation Metrics

We measure the outcomes using two primary metrics to isolate different dimensions of assertion quality:

- **Valid Runs:** The number of datapoints where the generated code successfully compiles, executes, and passes on the original unmutated program. This measures the usability and reliability of the generation process.
- **Test Strength:** The ratio of killed mutants strictly to the mutants covered by the test execution path, as detailed in Section 2.1. We calculate this average exclusively over the completed Valid Runs.

5 Results

We empirically evaluate the agentic workflow against both human developers and static LLM baselines. Table 1 merges the baseline and ablation results, displaying Valid Runs, Average Test Strength, and generation times.

5.1 Human Baseline vs. Static Generation

The static LLM achieved a Test Strength of 49.9%, but executed successfully in only 61 instances compared to the human baseline's 105.

5.2 Agentic Workflow Ablation

The refining loop is the largest contributor to overall valid runs. Static configurations complete 60 to 62 valid runs, while agentic configurations complete 82 to 89 valid runs. The loop raises average TS to approximately 56.0%.

The Summarizer paired with the Refining Loop achieves a Test Strength of 56.1% with 87 valid runs, taking 114.5 seconds on average. The pure Refining Loop completes 89 valid runs but takes 142.6 seconds.

The Planner alone improves static prompting slightly (from 49.9% to 51.2%). However, the Full Agent (Summarizer + Planner + Loop) completes 87 valid runs with TS 55.6%, taking the longest average runtime at 159.9 seconds.

Table 1: Performance comparison across human baseline, static generation, and agentic ablation configurations.

Configuration	Valid Runs	Avg Test Strength	Avg Generation time (s)
Human Baseline	105 (100%)	45.6%	24.0*
One Shot (Static)	61 (58.1%)	49.9%	46.0
Summarizer	60 (57.1%)	50.6%	50.3
Planner	61 (58.1%)	51.2%	63.1
Summarizer + Planner	62 (59.0%)	51.5%	55.9
Refining Loop	89 (84.8%)	56.0%	142.6
Planner + Loop	82 (78.1%)	55.7%	121.4
Summarizer + Loop	87 (82.9%)	56.1%	114.5
Summarizer + Planner + Loop (Full)	87 (82.9%)	55.6%	159.9

*For the human baseline, actual generation time is omitted; 24.0s represents the baseline framework execution overhead.

5.3 Discussion

In this subsection, we interpret the empirical results and manual observations.

The static baseline produces strong assertions when it succeeds, but we determine it is unreliable for practical oracle generation without execution feedback.

Regarding the ablation configurations, we interpret the Summarizer + Loop configuration as the optimal trade-off. It achieves the highest Test Strength and maintains high reliability without the computational overhead of the Full Agent. Because the Planner improved static generation but added little value inside the loop, the results suggest that execution feedback overrides the need for upfront strategic planning.

6 Threats to Validity

6.1 Internal Validity

The primary internal threat is data leakage. Because we utilize recent LLMs alongside established GitHub repositories, the models may have encountered these specific test suites during pre-training. However, both the static baseline and the agentic configurations use the exact same models, keeping the relative performance gains informative. We also use deterministic decoding with temperature 0 to ensure prompt-level reproducibility.

6.2 External Validity

Due to constraints, our dataset is limited to 112 focal tests from two repositories. We chose repositories with distinct structures to promote diversity, but this sample may not capture all software domains. Furthermore, our evaluation is restricted to Java. The agent's ability to self-correct relies partly on Java's strict compile-time errors, and these findings may not generalize identically to dynamically typed languages.

6.3 Construct and Statistical Validity

To mitigate construct threats, we utilize Test Strength rather than standard Mutation Score, isolating assertion quality from prefix reachability. We report aggregate averages and valid-run counts without statistical significance tests, meaning small Test Strength differences between agentic configurations require cautious interpretation.

7 Responsible Research and Data Availability

We adhere to the ethical use of computational resources. We constrain the ablation study to 112 focal tests and implement strict termination conditions.

To support open science, the complete agentic framework, dataset configurations, and execution scripts are publicly available. The repository contains the ablation runner, agentic workflow implementation, baseline generators, dataset mapping logic, and configuration files. We provide the full source code at https://github.com/horiagali/assertion-generator-java.

8 Conclusion

Automated generation of robust test oracles remains a primary bottleneck in software testing. While Large Language Models show potential for generating semantic assertions, static single-pass generation often produces uncompilable or failing code. We proposed and evaluated an agentic assertion generating workflow that embeds the LLM within an execution-grounded feedback loop.

Our ablation study on 112 focal tests from twilio-java and liqp shows that the Refining Loop is the main source of improvement. Static prompting completed only 58.1% of valid runs (61 instances) relative to the human baseline. In contrast, loop-enabled configurations completed up to 84.8% of valid runs (89 instances). This represents a 45.9% relative increase in reliable executions compared to static generation. Furthermore, the Summarizer + Refining Loop configuration achieves the highest average Test Strength at 56.1%. This delivers a 23.0% relative increase over the human baseline of 45.6%, and a 12.4% relative increase over the static configuration's 49.9% Test Strength, while running faster than the Full Agent. The Planner helped in static prompting but added little value inside the loop. These findings demonstrate that execution feedback is significantly more valuable for assertion quality than simply adding more upfront planning components.

9 Future Work

While our findings demonstrate the efficacy of agentic test generation, several avenues remain for future research. First, future studies

should evaluate this framework on a larger and more diverse dataset. Expanding the evaluation to include more repositories would help verify whether the observed loop and summarizer effects generalize across different software architectures and domains.

Second, researchers should investigate the application of this workflow to dynamically typed languages such as Python or JavaScript. Because our current framework heavily leverages strict compile-time errors in Java to guide self-correction, adapting the Critic node to rely on runtime execution failures will test the versatility of the agentic loop.

Future work should explore alternative context management strategies. The Summarizer + Loop result suggests that structured context can improve efficiency, but the best way to preserve syntax-relevant details remains open. Techniques such as targeted abstract syntax tree (AST) pruning could provide a better balance between context size and code fidelity.

References

[1] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing.* Cambridge University Press.

[2] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.

[3] Arda Celik and Qusay H. Mahmoud. 2026. GEM: A Framework for Strengthening LLM-Generated Unit Tests Using Mutation Feedback. In *Proceedings of the Ibero-American Conference on Software Engineering (CIbSE).*

[4] Bei Chu, Yang Feng, Kui Liu, Zhaoqiang Guo, Yichi Zhang, Hange Shi, Zifan Nan, and Baowen Xu. 2025. Large Language Models for Unit Test Generation: Achievements, Challenges, and Opportunities. *arXiv preprint arXiv:2511.21382* (2025).

[5] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: a practical mutation testing tool for Java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis.* 449–452.

[6] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering.* 416–419.

[7] Dong Huang, Mingzhe Du, Jie M. Zhang, Zheng Lin, Meng Luo, Qianru Zhang, and See-Kiong Ng. 2025. Nexus: Execution-Grounded Multi-Agent Test Oracle Synthesis. *arXiv preprint arXiv:2510.26423* (2025).

[8] S. M. Khandaker, F. Kifetew, D. Prandi, and A. Susi. 2025. AugmenTest: Enhancing Tests with LLM-Driven Oracles. *arXiv preprint arXiv:2501.17461* (2025).

[9] Davide Molinelli, Luca Di Grazia, Alberto Martin-Lopez, Michael D. Ernst, and Mauro Pezzè. 2025. Do LLMs Generate Useful Test Oracles? An Empirical Study with an Unbiased Dataset. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE).*

[10] Davide Molinelli, Alberto Martin-Lopez, Elliott Zackrone, Beyza Eken, Michael D. Ernst, and Mauro Pezzè. 2025. Tratto: A Neuro-Symbolic Approach to Deriving Axiomatic Test Oracles. *arXiv preprint arXiv:2504.04251* (2025).

[11] S. Primbs, B. Fein, and G. Fraser. 2025. AsserT5: Test Assertion Generation Using a Fine-Tuned Code Language Model. In *Proceedings of the IEEE/ACM International Conference on Automation of Software Test (AST).*

[12] Ravin Ravi, Dylan Bradshaw, Stefano Ruberto, Gunel Jahangirova, and Valerio Terragni. 2025. LLMLOOP: Improving LLM-Generated Code and Tests Through Automated Iterative Feedback Loops. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME).*

[13] Qinghua Xu, Guancheng splinter Wang, Lionel Briand, and Kui Liu. 2025. Hallucination to Consensus: Multi-Agent LLMs for End-to-End JUnit Test Generation.

A Agent Prompts and Templates

This appendix provides the exact system instructions and human prompt templates utilized by the reasoning nodes within the agentic workflow.

A.1 Summarizer Prompt

```
You are an expert Java mutation-testing semantic analyzer.

TASK:
Compress the provided Java project context into ONLY the
     information
required to generate high-quality mutation-killing
     assertions.

Your summary MUST preserve:
- executable test scope
- visible variables
- mutation-relevant APIs
- state propagation paths
- observable behaviors
- assertion opportunities
- branch-sensitive behaviors
- collection/map semantics
- null-handling semantics
- builder propagation semantics

OUTPUT FORMAT MUST BE EXACTLY:

TEST_SCOPE_VARIABLES:
- variable : type : origin

FOCAL_METHODS:
- methodSignature -> returnType

MUTATION_RELEVANT_APIS:
- ...

STATE_PROPAGATION:
- ...

ASSERTABLE_BEHAVIORS:
- ...

RETURN_SEMANTICS:
- ...

COLLECTION_SEMANTICS:
- ...

NULLABILITY:
- ...

MUTATION_HOTSPOTS:
- ...

CONSTRAINTS:
- ...

RULES:
1. ONLY summarize behavior explicitly derivable from the
     context.
2. NEVER invent APIs.
3. NEVER invent variables.
4. NEVER invent hidden state.
```

```
5. Preserve builder setter methods if they affect observable
      outputs.
6. Preserve methods relevant for surviving mutants.
7. Preserve APIs needed to construct meaningful object
     states.
8. Prefer semantic compression over implementation detail
     copying.
9. Omit irrelevant helper methods.
10. NO prose paragraphs.
11. NO markdown.
12. NO explanations outside bullet lists.
13. Do NOT infer builder/fluent APIs unless their signatures
      are present
    in the provided context.
```

A.2 Planner Prompt

```
You are a mutation testing strategist.

TASK:
Design assertion objectives that maximize PIT mutation score
     .

OUTPUT FORMAT MUST BE EXACTLY:

SAFE_EXISTING_VARIABLES:
- variable : type/origin : safe uses

SAFE_METHOD_CALLS:
- receiver.method(args) -> assertion-relevant return/effect

PRIMARY_TARGETS:
- ...

ASSERTION_CANDIDATES:
- assertion goal : exact variables/methods to use : expected
      value source

MUTATION_RISKS:
- ...

INVALID_ASSERTION_RISKS:
- ...

DO_NOT_USE:
- unavailable/speculative variable or API : reason

ASSERTION_PRIORITIES:
1. ...
2. ...
3. ...

RULES:
1. NO Java code.
2. NO assertions.
3. NO prose paragraphs.
4. Use compact bullet points only.
5. Focus on semantic verification.
6. Use ONLY variables and methods visible in the supplied
     context.
7. Flag invalid scope assumptions.
8. Prefer exact semantic checks over null checks.
9. Every assertion candidate must name the concrete existing
      receiver or
   constructor that the coder should use.
```

A.3 Coder Prompt

System Prompt:

```
You are an expert Java mutation-testing engineer.
```

Human Prompt Template:

```
{downstream_context_block(state)}

ASSERTION STRATEGY:
{improvement_plan or plan}

BEST GREEN ASSERTIONS SO FAR:
{best_prediction}

Use this as the safe base, not as an automatic final answer.
      Preserve its
correct assertions while repairing or replacing failed
     additions from the
last attempt so the next block still tries to improve on
     this best green
candidate.

LAST ATTEMPT:
{previous_code}

This is the most recent attempted assertion block. It may be
      worse than the
best green block above, and it may have caused the latest
     failure. Treat it
as a draft diff against the best green block: keep useful
     non-failing ideas,
but repair or replace failing lines.

LAST FAILURE / SANDBOX FEEDBACK:
{latest_feedback}

TASK:
Generate ONLY valid Java test-body code.
Write the COMPLETE, fully corrected assertion block.
If the last attempt caused a compilation or runtime failure,
REMOVE, FIX, or REPLACE only the failing lines.
The output replaces the entire previous assertion block.

REPAIR MODE:
If BEST GREEN ASSERTIONS SO FAR exists and LAST ATTEMPT
     failed:
1. Start from the best green assertions as the safe base.
```

```
2. Identify the exact failing additions from LAST FAILURE /
     SANDBOX FEEDBACK.
3. Do NOT simply return the best green block unchanged
     unless no safe
   improvement is possible.
4. Prefer replacing a failing assertion with a corrected
     assertion that
   targets the same mutant/behavior.
5. Preserve useful non-failing assertions from the last
     attempt when they
   are visible, scoped, and green-suite safe.
6. The final block should be at least as strong as the best
     green block
   while passing on the original implementation.

OUTPUT REQUIREMENTS:
1. Every semantic validation MUST use a JUnit assertion API.
2. NEVER output naked boolean expressions.
3. NEVER output passive method calls without assertions.
4. EVERY generated line must compile inside the target test
     body.
5. EVERY line must end with ';'
6. EVERY assertion must pass on the original unmutated
     implementation.
7. If feedback reports BASELINE TEST FAILURE, prioritize
     green-suite
   correctness over mutation score.

ALLOWED ASSERTION APIS:
- assertEquals(...)
- assertNotEquals(...)
- assertTrue(...)
- assertFalse(...)
- assertNotNull(...)
- assertNull(...)
- assertSame(...)
- assertNotSame(...)
- assertArrayEquals(...)
- fail(...)

ALLOWED CODE:
- local variable declarations
- inline object construction
- method calls
- collection/map inspection
- intermediate values required for assertions

FORBIDDEN CODE:
- naked expressions like:
  x != null;
  foo.isEmpty();
  a == b;

- helper methods
- classes
- imports
- packages
- annotations
- markdown
- prose
- comments
- explanation text
- method wrappers

SCOPE RULES:
1. Use ONLY identifiers explicitly visible in:
   - TARGET TEST METHOD
   - TEST CLASS FIELDS
   - SETUP / TEARDOWN METHODS
   - TEST CLASS HELPER METHODS
   - visible focal constructors
   - visible focal methods
   - visible constants
   - visible fields
   - ASSERTION MANIFEST, if present

2. NEVER invent:
   - mocks
   - services
   - factories
   - builders not explicitly visible
   - hidden state
   - undeclared variables
   - unavailable APIs

3. NEVER call instance methods without a receiver object.

4. If no reusable variables are visible in the target test
     method or
   test class fields:
   - instantiate objects inline using ONLY visible
     constructors
   - prefer inline constructor expressions over temporary
     variables

5. You MAY instantiate ONLY classes whose constructors are
     explicitly visible.

ASSERTION QUALITY RULES:
1. Prefer semantic assertions over trivial assertions.
2. Prefer exact value verification over null checks.
3. Prefer behavioral verification over existence checks.
4. Prefer mutation-killing assertions over broad assertions.
5. Avoid duplicate assertions.
6. Avoid redundant assertions.
7. Avoid weak assertions that always pass.
8. Avoid object-wide toString() or broad string contains
     assertions
   unless the focal behavior is explicitly a toString
     contract and the
   exact expected string is visible in the context.
9. Prefer dedicated getters, fields, collections, maps, or
     returned values
   over indirect debug/string representations.
```

```
10. Never infer expected values only from the test method
     name.

MUTATION-GUIDED RULES:
1. Focus assertions on surviving mutant behavior.
2. Strengthen validation around:
   - return values
   - conditionals
   - field propagation
   - constructor effects
   - collection contents
   - null handling

3. If surviving mutants involve:
   - ReturnVals mutators:
       validate exact returned values

   - Conditional mutators:
       validate branch-sensitive behavior

   - MemberVariable mutators:
       validate propagated state and attributes

   - Null mutators:
       validate null-sensitive semantics

OUTPUT FORMAT:
- Output ONLY raw Java test-body statements.
- No surrounding text.
- No markdown fences.
- No explanations.

If uncertain, output fewer but stronger assertions.
```

A.4 Critic Feedback Templates

The Critic is a deterministic script that populates the `{latest_feedback}` variable using the following formats:

Compile Failure:

```
COMPILATION FAILURE

RAW COMPILE ERROR:
{compile_error}
```

Green-suite Failure:

```
BASELINE TEST FAILURE

The assertion block compiled, but failed on the original
     unmutated code.
The next revision must remove or fix the failing assertion
     before trying
to kill more mutants.

RAW TEST FAILURE:
{test_failure}
```

PIT Execution Failure:

```
PIT EXECUTION FAILURE

RAW PIT FAILURE:
{pit_feedback}
```

Successful PIT Run:

```
TEST STRENGTH: {test_strength}
MUTATION SCORE: {mutation_score}
SURVIVING MUTANTS:
{mutant_details}
```

A.5 Improver Prompt

System Prompt:

```
You are a senior Java mutation-testing reviewer.

TASK:
Analyze why the previous assertion set failed or was weak.

OUTPUT FORMAT MUST BE EXACTLY:

FAILURE_ANALYSIS:
- ...

SURVIVING_MUTANT_ANALYSIS:
- ...

MISSING_VALIDATIONS:
- ...

INVALID_SCOPE_USAGE:
- ...

EXACT_LINES_TO_REMOVE_OR_FIX:
- previous assertion/local statement : reason

REPAIR_STRATEGY:
- how to replace each failing line with a green-suite-safe
     assertion
- which non-failing additions from the last attempt should
     be kept
- how the repaired block can still improve over the best
     green block

NEXT_ASSERTION_GOALS: (keep in mind that we can only write
     assertions,
not write any new tests or modify any other code)
- ...

RULES:
1. NO Java code.
2. NO prose paragraphs.
3. NO explanations outside bullet points.
4. Focus on semantic gaps.
5. Focus on scope errors.
6. Focus on compile failures.
7. Focus on surviving mutant causes.
```

```
8. If feedback says BASELINE TEST FAILURE:
   - treat the previous assertions as invalid on the
     original code
   - identify the exact failing assertion or exception from
     RAW TEST FAILURE
   - recommend correcting or replacing failing lines before
     adding strength
   - do not recommend simply reverting to the best green
     block unless no
     safe replacement exists
   - focus on preserving non-failing improvements from the
     last attempt
9. Name concrete variables/method calls from the context
     whenever proposing
   a next assertion goal.
10. NO Java code, but name the intended assertion semantics
     precisely.
```

Human Prompt Template:

```
{downstream_context_block(state)}

PREVIOUS ASSERTIONS:
{prediction}

EXECUTION FEEDBACK:
{latest_feedback}
```